EXHIBIT G

From:	Brian Swetland.	Sent:8/16/2006 4:40 AM.
To: [-]	fadden@google.com; ficus@google.com; cjd@google.com.	
Cc: [-]	.	
Bcc: [-]	.	
Subject:	feedback welcome.	

The following is a document I'm calling DESIGN_MANIFESTO at the moment.
Does this stuff make sense? Am I smoking the crack?

Brian

- architecture: priority one is the device
- we are shipping the device, not the simulator
- if the device is not fast and stable we FAIL
- the emulator is the answer for desktop work
- yes, it is slow: we must make it faster
(the end users will never judge us by how fast the simulator was)
- performance is a problem NOW not LATER
- fadden's exception: allow it to build on linux-x86 for valgrinding
(brian's note: it need not be pretty here)

- abandon single process support
- we are shipping multiprocess, not single process
- extra code to allow us to operate in two modes MUST GO
- hard to debug? we must write better tools
- again we must build and debug what we will ship

- platform: priority one is user experience
- if we do not ship a compelling experience
(dialer, pim, maps, whatever) we FAIL
- the platform must serve the apps & experience
- writing great apps must be simple
- "it is complicated because it is powerful" is a lousy answer
to "why is it so hard to do X"

- build on top of standard linux kernel services
- avoids making the kernel bigger
(can't remove core services: let's use em!)
- relies on well tested existing services
- in particular:
- unix domain sockets (dgram and stream)
- make use of the private linux-domain namespace
- shm passing using fd-over-socket
- rights checking using privs-over-socket

- avoid userland unix-ism
- we are an embedded system, not unix-on-a-phone
- do not keep state in a billion textfiles
- do not rely on the shell for anything besides debugging

- write it once
- never use two APIs or systems where one will do
- avoid excessive layering: in the long run, this overhead kills
- only one object model
- use the java object model
- do not build elaborate c++ object models

- minimal native code
- write as much as possible in java
- we are building a java based system: that decision is final

- java is easier to debug (modulo vm stability)
I'm serious here: GDB vs Java IDE Debuggers - think about it
- java is denser
- java is safer (bounds checking, perms, etc)
- favor C over C++ for native glue
- keep it light and fast
- keep it simple and clean
- smaller is faster
- write it in java first (barring certain special cases)
- we can always move java to native to make things faster

- threading model
- use as few as you can
- favor libevent/select for muxing

- Java not Sun
- avoid excessive koolaid drinking
- we are building a java based system, not pushing Sun's agendas
- remember that we are CDC/MIDP, not J2SE
- allocate sparingly
- keep it small
- avoid reflection: it is very costly

- notes on design for embedded linux
- use unix domain sockets for local communication (secure!)
- use tcp sockets for debugging connections
(make sure we do not accept such connections over wireless network)
- use credential passing/checking to further restrict access if need be
- do *not* use sysV shm, sems, etc
- avoid using signals if at all possible
- use unix domain sockets to determine if a remote process went away
(when the far side closes the local side should close: verify)
- only the init service should start/stop processes
(expand it so it can queue up one-shot processes like vm instances)
- use inotify to monitor file
- perhaps we should always use libevent for muxing?